MASSACHUSETTS INSTITUTE OF TECHNOLOGY

VLSI PUBLICATIONS

VLSI Memo No. 88-491
December 1988

The J-Machine: System Support for Actors

William J. Dally

Abstract

The J-Machine in concert with its operating system kernel, JOSS, provides low-overhead system services to support actor programming systems. The J-Machine is not specialized to actor systems; instead, it provides primitive mechanisms for communication, synchronization, and translation. Communication mechanisms are provided that permit a node to send a message to any other node in the machine in < 2μs. On message arrival, a task is created and dispatched in < 1μs. A translation mechanism supports a global virtual address space. These mechanisms efficiently support most proposed models of concurrent computation. The hardware is an ensemble of up to 65,536 nodes each containing a 36-bit processor, 4K 36-bit words of memory, and a router. The nodes are connected by a high-speed 3-D mesh network. This design was chosen to make the most efficient use of available chip and board area.

Microsystems Research Center, Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138

Acknowledgements

This work was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-87-K-0825 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.

Author Information

Dally: Artificial Intelligence Laboratory and Laboratory of Computer Science, MIT, Room NE43-417, Cambridge, MA 02139. (617) 253-6043, billd@wheaties.ai.mit.edu.

The J-Machine: System Support for Actors [1]

William J. Dally
Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

The J-Machine in concert with its operating system kernel, JOSS, provides low-overhead system services to support actor programming systems. The J-Machine is not specialized to actor systems; instead, it provides primitive mechanisms for communication, synchronization, and translation. Communication mechanisms are provided that permit a node to send a message to any other node in the machine in $< 2\mu s$. On message arrival, a task is created and dispatched in $< 1\mu s$. A translation mechanism supports a global virtual address space. These mechanisms efficiently support most proposed models of concurrent computation. The hardware is an ensemble of up to 65,536 nodes each containing a 36-bit processor, 4K 36-bit words of memory, and a router. The nodes are connected by a high-speed 3-D mesh network. This design was chosen to make the most efficient use of available chip and board area.

1 Introduction

Overview

The J-Machine is a distributed-memory, MIMD, concurrent computer. In concert with its operating system kernel, JOSS, the J-Machine provides low-overhead system services to support actor programming systems. The combined hardware/software system efficiently implements two abstractions: object and task. An object is a named collection of data. All data in the system: program data, code, and contexts are objects. The object namespace is global – an object can be referenced from any node of the machine. Tasks are procedure activations that may operate on the state of objects. To support fine-grain concurrent programming systems, the system is designed to handle small objects (8 words) and small tasks (20 instructions).

The J-Machine is not specialized to actor systems; instead, it provides primitive mechanisms for communication, synchronization, and translation. Communication mechanisms are provided that permit a node to send a message to any other node in the machine in $< 2\mu s$. No processing resources on intermediate nodes are consumed and buffer memory is automatically allocated on the receiving node. The synchronization mechanisms schedule and dispatch a task in $< 1\mu s$ on

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-87-K-0825 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and International Business Machines Corporation.

message arrival and suspend tasks that attempt to reference data that is not yet available. The translation mechanism maintains bindings between arbitrary names and values. It is used to perform address translation to support a global virtual address space. These mechanisms have been selected to be both general and amenable to efficient hardware implementation. They efficiently support many parallel models of computation including actors[1], dataflow[17], and communicating processes[21].

The hardware is an ensemble of up to 65,536 message-driven processors (MDPs)[9]. This limit is set by the addressability of the router and the bandwidth of the network. Each node contains a 36-bit processor, 4K 36-bit words of memory, and a communications controller (router). The nodes are connected by a high-speed, three-dimensional mesh network. Each network channel has a bandwidth of 450Mbits/s. The first medium-scale prototype will be a 4096-node system.

This design was chosen to make the most efficient use of available chip and board area. Packaging a small amount of memory on each node gives us an extremely high memory bandwidth (3Gbits/s per chip or 200Tbits/s in a fully populated system). Memory consumes most of the chip area; from one point of view, the system is a memory with processors added to each node to improve bandwidth for local operations. The fast communication and global address space prevent the small local memories from limiting programmability or performance. The 3-D network gives the highest throughput and lowest latency for a given wire density[7][14]. It allows the processing nodes to be packed densely and results in uniformly short wires.

The J-Machine project is driven by the following goals:

- To identify and implement simple hardware mechanisms for communication, synchronization, and naming suitable for supporting a broad range of concurrent programming models.
- To reduce the overhead associated with these mechanisms to a few instruction times so that fine-grain programs may be efficiently executed.
- To design an area-efficient machine: one that maximizes performance for a given amount of chip and wiring area.

Grain Size

The J-Machine is a fine-grain concurrent computer in that (1) it is designed to efficiently support fine-grain programs and (2) it is composed of fine-grain processing nodes [15].

The *grain size* of a program refers to the size of the tasks and messages that make up the program. Coarse-grain programs have a few long ($\approx 10^5$ instruction) tasks, while fine-grain programs have many short ($\approx$ 20 instruction) tasks. With more tasks that can execute at a given time – viz. more concurrency – fine-grain programs (in the absence of overhead) result in faster solutions than coarse-grain programs.

The *grain size* of a machine refers to the physical size and the amount of memory in one processing node. A coarse-grain processing node requires hundreds of chips (several boards)

and has $\approx 10^7$ bytes of memory while fine-grain node fits on a single chip and has $\approx 10^4$ bytes of memory. Fine-grain nodes cost less and have less memory than coarse-grain nodes, however, because so little silicon area is required to build a fast processor, they need not have slower processors than coarse-grain nodes.

Background

The J-Machine builds on previous work in the design of message-passing and shared memory machines. Like the Caltech Cosmic Cube [33], the Intel iPSC [23], and the N-CUBE [29], each node of the J-Machine has a local memory and communicates with other nodes by passing messages. Because of its low overhead, the J-Machine can exploit concurrency at a much finer grain than these early message passing computers. Delivering a message and dispatching a task in response to the message arrival takes $< 3\mu s$ on the J-Machine as opposed to 5ms on an iPSC. Like the BBN Butterfly [4] and the IBM RP3 [30] the J-Machine provides a global virtual address space. The same IDs (virtual addresses) are used to reference on and off node objects. Like the InMOS transputer [22] and the Caltech MOSAIC [27] a J-Machine node is a single chip processing element integrating a processor, memory, and a communication unit.

The J-Machine is unique in that it extends these previous efforts with efficient primitive mechanisms for communication, synchronization and naming.

Summary

The remainder of this paper describes the J-Machine and the JOSS operating system kernel. Section 2 gives an overview of the system describing how the network, processor, and operating system layers work together to provide services. The network is described in Section 3. The topology, performance, and router design are discussed. Section 4 deals with the message driven processor and the mechanisms it provides for concurrency. The operating system is briefly described in Section 5.

2 System Architecture

The J-Machine system is layered as shown in Figure 1. A hardware layer provides primitive mechanisms. A software layer, the JOSS kernel, uses these primitives to build system services in two stages. First, local services are provided on each node. Second, global services are built on top of these local services by passing messages between the nodes.

The hardware layer consists of an ensemble of (up to 64K) MDP-based processing nodes connected by a three-dimensional mesh network. The network delivers messages between nodes. All routing and flow-control are handled in the network. The network consists of a communication controller or router on each node, and wires connecting these routers to their neighbors in each of the three physical dimensions.

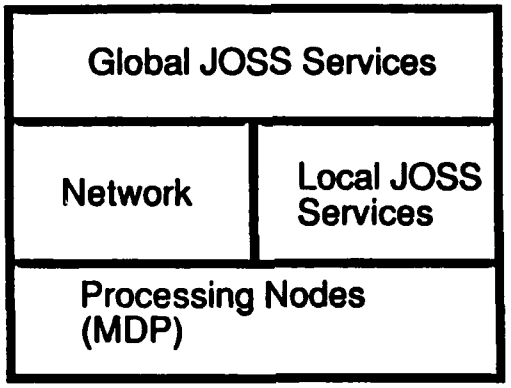

Figure 1: The J-Machine System. The hardware layer consists of a number of MDPs connected by a network. The operating system kernel running on each node provides local services. Global services are provided by exchanging messages with other nodes.

Each node contains a memory, a processor, and a communication controller. The communication controller is logically part of the network but physically part of the node. The 4K-word by 36-bit memory is used to store objects and system tables. Each word of the memory contains a 32-bit data item and a 4-bit tag. In addition to the usual uses of tags to support dynamic typing and garbage collection, special tags are provided to synchronize on data presence and to indicate if an address is local or remote. Memory accesses to write messages or read code are made a row (144bits) at a time to improve memory bandwidth. A part of the memory can be mapped as a set associative cache. This cache is used to implement the processor's translation mechanism.

The processor is message driven. It executes user and system code in response to messages arriving over the network. A conventional processor is instruction-driven in that it fetches an instruction from memory and dispatches a control sequence to execute the instruction. A message-driven processor receives a message from the network and dispatches a control sequence to execute the task specified by the message. The MDP uses an instruction sequence to *execute* a message. Hardware mechanisms for communication, synchronization, and translation are provided to accelerate the dispatch operation and the subsequent task execution.

To support communication over the network, the MDP provides a `SEND` instruction and performs automatic buffering of arriving messages. To synchronize execution with arriving messages, a primitive dispatch operation is provided that eliminates scheduling overhead. To synchronize on data, tags are provided to support futures. A general translation mechanism uses a set associative cache in the node memory to maintain arbitrary bindings.

The software layer is implemented by the JOSS kernel [36] which provides services for allocating memory and processor resources to objects and tasks. The first level of JOSS provides memory management and multitasking on each node independently. Objects are allocated locally as memory segments and assigned a unique name. A binding is maintained between the object name (its ID or virtual address) and the base and length of its segment. Each node allocates

names from a different partition of the global name space so the local object name is also its global name.

The second level of JOSS extends services globally across the network. A distributed global name table is maintained that contains bindings of object names to node addresses (node numbers). Remote objects are referenced by translating the object name to a node number and sending a message containing the object name to the node. At the node, the name is translated into a local segment descriptor. Objects are free to migrate from node to node. The global name table keeps track of the object's current location.

To handle large objects, JOSS provides support for *distributed objects* [5]. A distributed object is a collection of objects distributed over many nodes that share a common name. JOSS translates distributed object names to the location of the nearest *constituent object.*

The JOSS task manager schedules tasks in response to message arrival. It makes heavy use of the MDP synchronization mechanisms. The MDP dispatch mechanism is used to create tasks in response to message arrival. Tasks that terminate without suspending require no further services. If a task suspends awaiting a message, a context object is created to hold its state. When the message to restart the task arrives, its context is reloaded to resume the task.

To see how the system functions together, consider the example shown in Figure 2. In Figure 2a, A task executing in Context 37 on Node 124 sends a `Sum` message to an object, `point1`. This message requests that the object sum its two fields `x` and `y`. The sending node translates the object name for `point1` (a unique 32-bit pattern) into a node address, Node 262 (a 16-bit integer), using the MDP translation mechanism. A sequence of MDP `SEND` instructions is then used to inject the message into the network. The message includes (1) the node address of `point1` (Node 262), (2) the object name of `point1`, (3) the message name or selector, `Sum`, and (4) a continuation (the node and ID of the sender's context and the slot into which the reply should be stored). The sending task continues to execute until it needs the result of the `Sum` message.

The network delivers the injected message to Node 262. At this node, the MDP buffering mechanism allocates storage for the message and sequences the message off the network into the node's memory. When the node completes its current task (and any other tasks ahead in the queue), the MDP dispatch mechanism creates a new task in response to the message. This task translates the ID of `point1` into a segment descriptor for the object, adds the `x` and `y` fields of the object together, and uses a sequence of `SEND` instructions to inject a message containing the sum into the network. As shown in Figure 2b, this message contains (1) the node address of the sender's context, (2) the ID of the sender's context, (3) the context slot awaiting the reply, and (4) the result. This task then terminates.

The network delivers the reply message to Node 124 where it is buffered and eventually dispatched to create a task. This task translates the ID for Context 37 into a segment descriptor. The reply value is stored into the specified slot of this context. The sending task is then resumed by loading its context from this segment.

The round trip delay for this example message send and reply is $\approx 5\mu s$. The difficulty in building a concurrent system the scale of the J-machine is not developing the mechanisms conceptually.

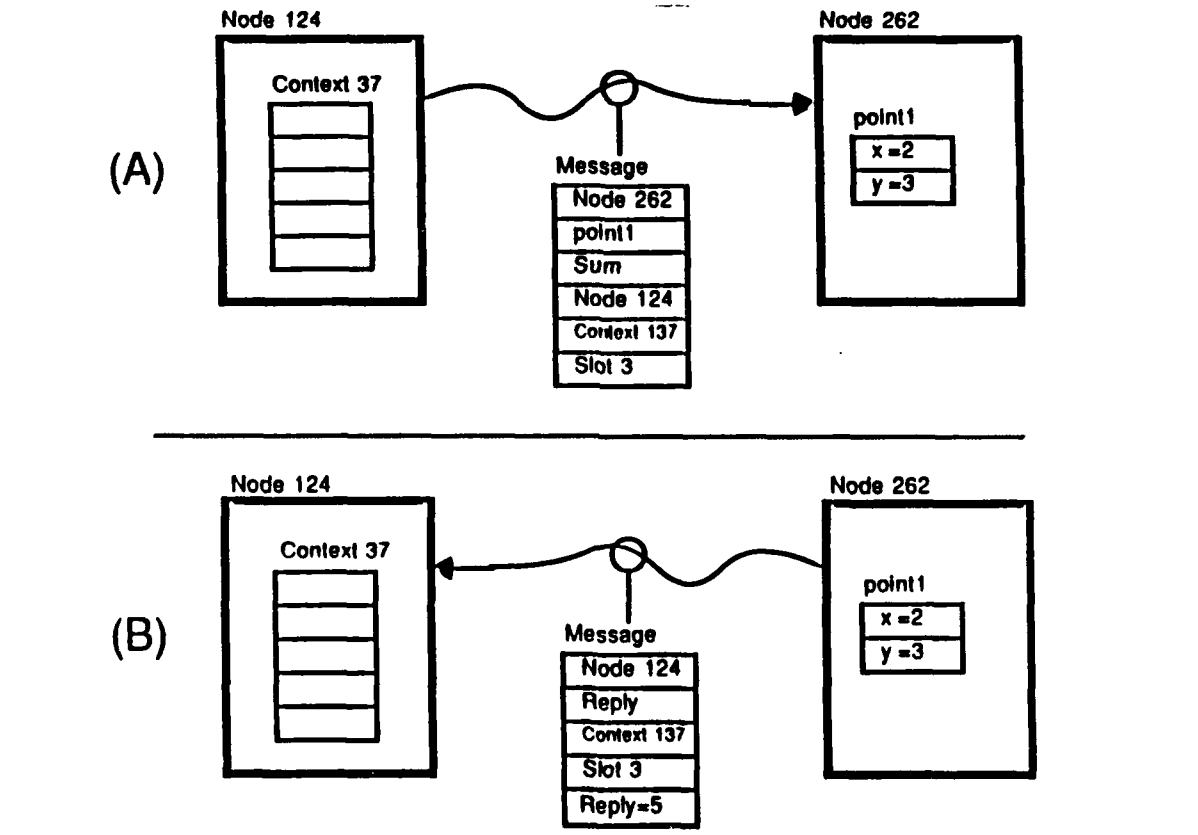

Figure 2: (a) A task executing in Context 37 on Node 124 sends a message to object `point1` requesting that it perform the `Sum` method. (b) A reply message is returned to Context 37.

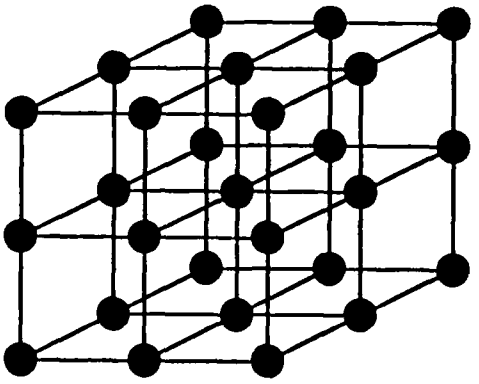

Figure 3: The J-Machine Network is a 3-D mesh or k-ary 3-cube (a 3× 3×3 mesh is shown here). Messages injected into the network at any node are routed to the destination node specified in the head of the message. All routing and flow control is performed in the network.

It is implementing them efficiently so the overhead of accessing remote nodes is made small enough to permit the execution of fine-grain programs.

In the following sections we will examine the implementation of each component of the J-Machine system.

3 The Network

The J-Machine network has a 3-D mesh topology as shown in Figure 3. Each node is located by a three coordinate address (x, y, and z). A node is connected to its six neighbors (if they exist) that have addresses differing in only one coordinate by ± 1. All connections are bidirectional channels. Each channel requires 15 wires to carry 9-data bits, one tail bit, and five control lines [10]. Addressing is provided to support up to a $32 \times 32 \times 64$ cube of 65536 nodes. The prototype will be built as a $16 \times 16 \times 16$ cube of 4096 nodes. For a machine, such as the J-Machine where wire density is a limiting factor, this topology has been shown to give the lowest latency and highest throughput for a given wire density [7][14].

The network topology is not visible to the programmer. The latency of sending a message from any node, i, to any other node, j, is sufficiently low that the programmer sees the network as a complete connection. Zero load network latency is given by

$$T = T_d D + T_c \frac{L}{W}. \quad (1)$$

Where D is the distance (number of hops) the message must travel, L is the length of the message in bits, and W is the width of the channel in bits. The network is expected to have a propagation delay per stage, T_d, of 20ns and a channel cycle time, T_c of 20ns. With these times, a six word (L =216 bit) message traversing half the network diameter (D = 24) has a latency of

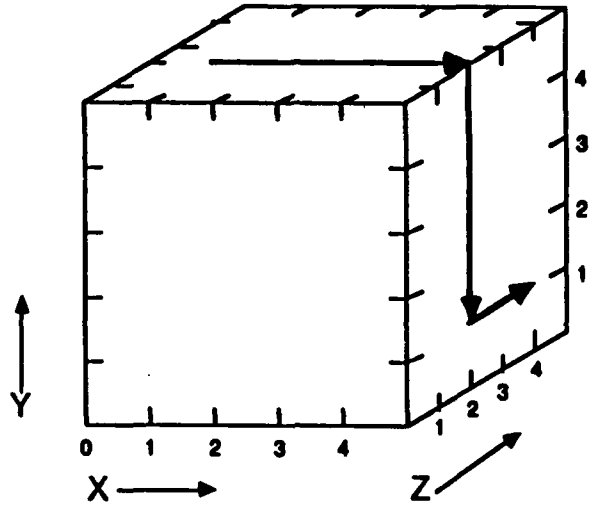

Figure 4: The J-Machine Network performs e-cube or destination tag routing. Messages are routed in each dimension in turn to the proper coordinate in that dimension. In this figure a message is routed from (1,5,2) to (5,1,4) routing first in x, then y, then z.

960ns equally divided between the two components of latency [14]. An average message travels one third of the network diameter for a latency of 800ns.

The network provides all end to end message delivery services. The sending node injects a message containing the absolute address of the destination node. The network determines the route of the message, and sequences each flit (flow-control digit) of the message over the route. Flow control is performed as required to resolve contention and match channel rates.

There is no acknowledgement, error detection, or error correction on the network channels. The network wires are all short, contained within a single physical cabinet, and operated at low impedance. The error rate of a network channel is no higher than that of a properly terminated signal in a conventional CPU.

The J-Machine network uses e-cube routing, a deterministic routing algorithm. Messages are routed one dimension at a time as illustrated in Figure 4. At the sending node, the source address is subtracted from the absolute destination address to yield a relative destination address. The three coordinates of the relative destination address are contained in the three leading flits of the message. Routing is performed according to the relative address one coordinate at a time. After each hop the leading coordinate is adjusted to reflect the current position of the head of the message. When this coordinate reaches zero, routing is complete in that dimension. The coordinate is then stripped off and routing begins in the next dimension.

In Figure 4, the relative address is (4,-4,2). The message is first routed four hops in the positive x direction. When the relative address reaches (0,-4,2) at node (5,5,2) the message is at the

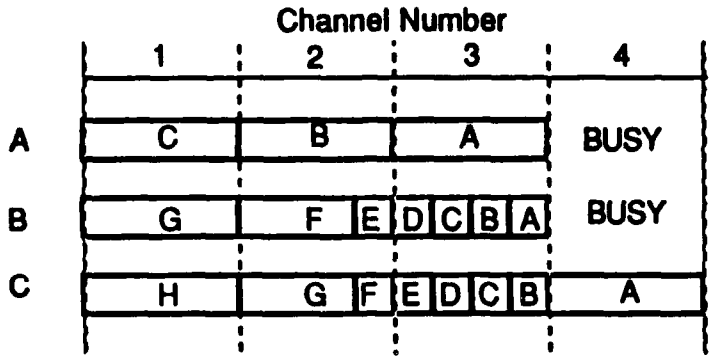

Figure 5: The J-Machine Network performs blocking flow control with four stages of queueing per node. (a) Message arrives at busy channel. (b) Message becomes compressed by queueing. (c) Channel is available; message continues advancing.

proper x coordinate. The x address is stripped off the message header and the message is routed four hops in the negative y direction to bring it to the proper y coordinate at (5,1,2). Two hops in the positive z direction brings the message to its destination.

To support system services operating at different priorities, two logical networks are provided, one for each priority level. The two logical networks have completely separate buffers and control state, but share the same set of physical channels between nodes. Each level does see the presence of the other except when performance degrades because a physical channel is being shared. If one network becomes completely congested, the other network still functions normally [10].

The network performs blocking flow control to resolve contention as shown in Figure 5. When a message requires use of a channel that is busy, it is blocked. The head of the message stops and begins filling the current channel's four-stage queue. When the four stages of queuing are full, the blockage propagates back to the preceding channel. Finally, when the channel becomes available, the message continues to advance toward its destination. This flow control is performed in a manner that is provably deadlock free[8].

While logical channels are allocated on a message by message basis, the bidirectional wires between two nodes (shared by four logical channels) are allocated to messages at different priorities and/or traveling in opposite directions on a flit-by-flit basis. Thus contention for the bidirectional channel slows messages without blocking them.

The internal structure of a network router is illustrated in Figure 6. The router consists of three levels. At the highest level, two completely separate priority routers interact only by competing for access to the physical communication channel (Figure 6a). Each priority consists of six dimension routers that handle routing in the positive and negative x,y, and z dimensions (Figure 6b). As described above, a message routes in a single dimension until it reaches the proper coordinate. The message then enters the next dimension in sequence.

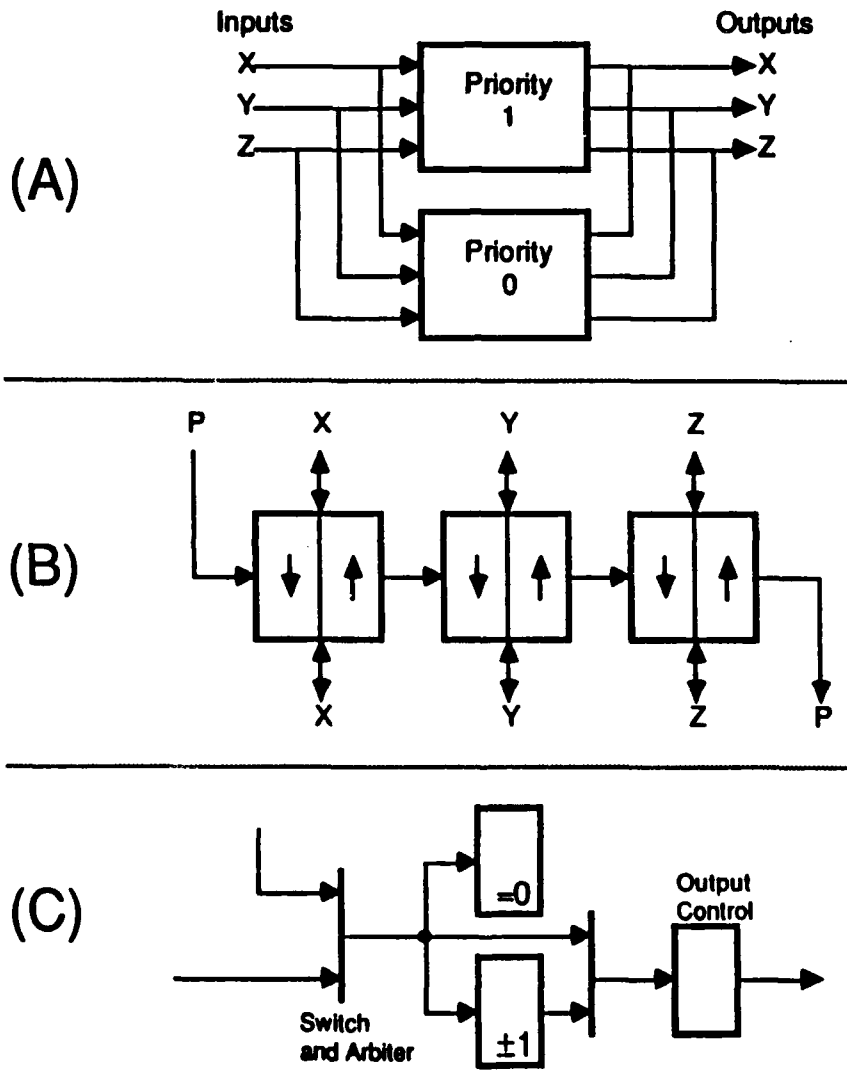

Figure 6: Block diagram of a router. (a) The two priorities are completely separate except where they share the physical channels. (b) Each priority contains three dimension data paths. (c) Each dimension data path performs switching and flow control for one dimension of routing.

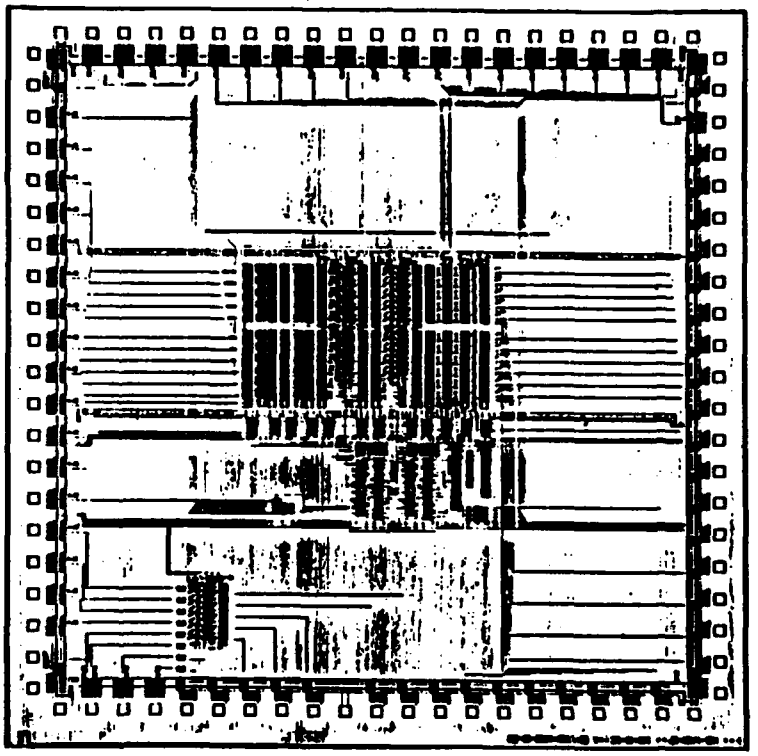

Figure 7: Photomicrograph of the network design frame, a prototype of the J-Machine router.

Each dimension router (Figure 6c)consists of an input switch, a zero checker, a decrementer, and an output controller. Messages entering the dimension compete with messages continuing in the dimension at a 2 to 1 switch. Once one message is granted this switch, the other input is locked out for the duration of the message. Once a message passes the input switch, it is zero checked and its head (routing) flit is decremented. If the head flit is non-zero the message continues in the current direction. Otherwise the head flit is stripped and the message is routed to the next dimension. Small (four flit) output buffers are provided on both outputs. Once the head flit of the message has set up the route, subsequent flits follow from the input switch directly to the output buffer, bypassing the decrementer. Figure 7 shows a prototype routing chip that implements two dimensions of the logic shown in Figure 6.

The routers are completely self-timed. There is no global clock. Arbiters are used at all switches where two event streams merge to avoid synchronization errors.

4 The Message-Driven Processor

The message-driven processor (MDP) is a 36-bit single-chip microcomputer specialized to operate efficiently in a multicomputer. [9][13]. The MDP chip includes the processor, a 4K-word

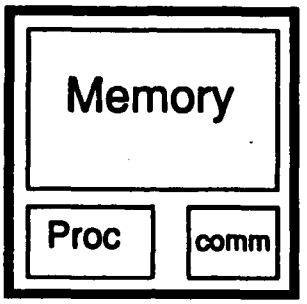

Figure 8: The Message-Driven Processor chip incorporates a 36-bit processor, a 4K-word x 36-bit memory, and a router (described above).

by 36-bit memory, and a router (Figure 8). An on-chip memory controller with ECC permits local memory to be expanded up to 1M-words by adding external DRAM chips.

Other machines have combined processor, memory, and communications on a single chip [22] [27] [29]. The MDP extends this work by providing fast, primitive mechanisms for synchronization, communication, and translation (naming) that allow the processor to efficiently support many parallel models of computation. A fast network is of little use if very large overheads are required to initiate and receive messages at the processing nodes. The MDP's mechanisms reduce the overhead of interacting with other processors over the network to levels that make fine-grain parallelism efficient.

The following mechanisms are provided:

- **Communication Mechanisms**
  - A `SEND` instruction injects messages into the network.
  - Messages arriving from the network are automatically buffered in a circular queue.
- **Synchronization Mechanisms**
  - A dispatch mechanism creates and schedules a task (thread of control and addressing environment) to handle each arriving message.
  - Tags for *futures* [19] synchronize tasks based on data dependencies.
- **Translation**
  - `ENTER` and `XLATE` (translate) instructions make bindings between arbitrary 36-bit key and data values (`ENTER`) and retrieve a value given the corresponding key (`XLATE`).
  - Segmented memory management provides relocation and protection for data objects stored in a node's memory.

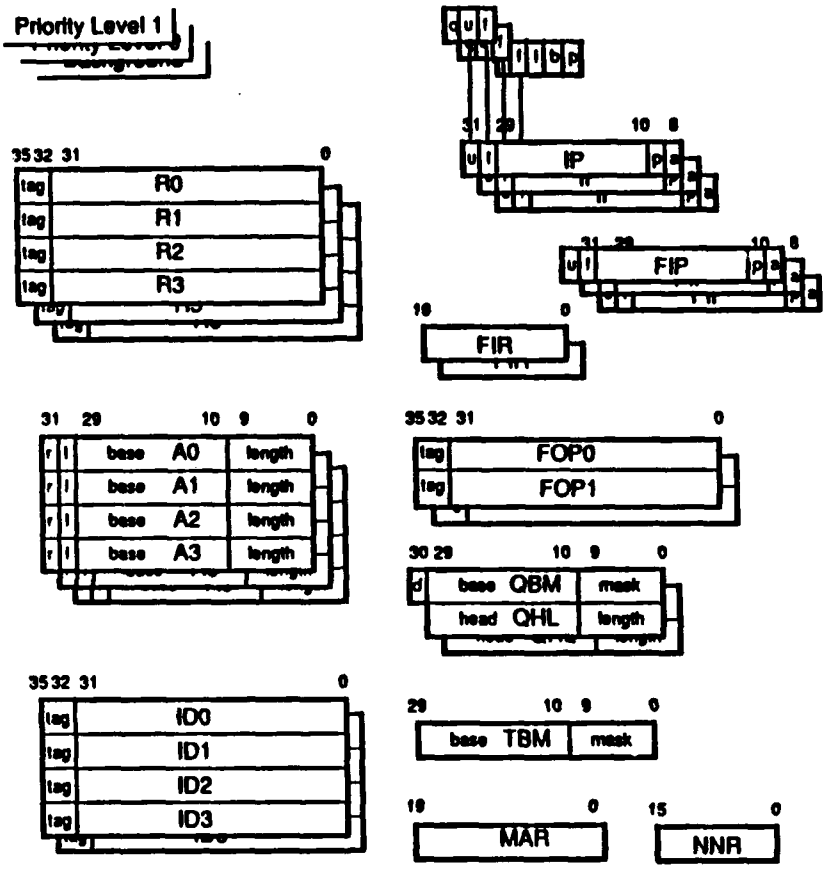

Figure 9: The MDP machine state contains three register sets, one for each of two message priorities and one for background execution.

The processor is *message driven* in the sense that processing is performed in response to messages (via the dispatch mechanism). There is no receive instruction. A task is created for each arriving message to handle that message. A computation is advanced (driven) by the messages carrying tasks about the network.

4.1 User Architecture

This section gives a brief overview of the user architecture. For a complete description, the user should consult [13].

Processor State

Figure 9 shows the register set of the MDP. There are three copies of most registers. One copy holds the state of the task being executed in response to the most recent priority zero message,

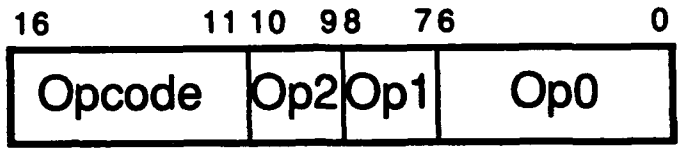

Figure 10: The MDP instruction format. Instructions are three address with two of the operands restricted to the general registers.

a second set handles priority one tasks, and the third set holds the state of the background task that executes when the node is awaiting a message. The three register sets enable the MDP to task switch between priorities without saving or restoring state.

Each priority contains four general registers, R0-R3, that can be used to hold arbitrary data. The register set is kept small to minimize the time required to save state when suspending a task. Also, the on-chip memory can be accessed in a single clock cycle. Fast local memory eliminates many of the advantages of large register sets.

Four segment registers, A0-A3, hold segment descriptors for the current addressing environment of a task. Each descriptor contains the base of the segment in local memory and its length. If the i-bit of a descriptor is set, the segment is invalid. The r-bit indicates if the segment is relocatable. The i and r bits are used to invalidate relocatable segments when the heap is compacted. The four ID registers, ID0-ID3, hold the names (virtual addresses) of the segments. The MDP translation mechanism is used to convert segment names into segment descriptors.

The instruction pointer, IP, locates the current instruction in the code segment (A0). Included in the IP are three status bits, U, F, and A . If the U (unchecked) bit is set, no type checking is performed. The F bit indicates when the machine can handle faults. A fault occuring when the F bit is set results in an unrecoverable double fault. If the A (absolute) bit is set, the IP is used as an absolute address and not as an offset into the code segment. These bits are included in the IP so the control state of the machine can be saved or restored by storing or loading a single register.

The FIP, FIR, and FOP registers are used for fault handling. When a fault or system call occurs, these registers are loaded from the current machine state. The fault handler examines the registers to correct the fault and returns from the fault by moving FIP into IP.

The QBM and QHL registers determine the memory allocated to the message queue, and the current state of the queue. The TBM register determines the memory allocated to the translation buffer.

Instruction Set

The MDP uses three address instructions as shown in Figure 10. Two of the operands are restricted to be general registers (R0-R3). The remaining operand can be any register, or a

memory location specified by a displacement or index into one of the segments. This instruction format was selected to give good code density – important with a small local memory – while permitting efficient access to variables stored in memory.

A synopsis of the instruction set is shown in Figure 11. In addition to the usual instructions for data movement, arithmetic, and control, the MDP provides instructions for sending messages over the network and for performing associative translations. If the U bit is clear, all instructions are type checked, the figure shows the admissible types for each instruction. This run-time type checking in combination with fast fault handling efficiently supports both dynamic typing and suspension of tasks when undetermined futures are touched.

4.2 Send Instruction

The MDP injects messages into the network using a send instruction that transmits one or two words (at most one from memory) and optionally terminates the message. The first word of the message is interpreted by the network as an absolute node address (in x,y format) and is stripped off before delivery. The remainder of the message is transmitted without modification. A typical message send is shown in Figure 12. The first instruction sends the absolute address of the destination node (contained in R0). The second instruction sends two words of data (from R1 and R2). The final instruction sends two additional words of data, one from R3, and one from memory. The use of the SENDE instruction marks the end of the message and causes it to be transmitted into the network. In a Concurrent Smalltalk message [16], the first word is a message header, the second specifies the receiver, the third word is the selector, subsequent words contain arguments, and the final word is a continuation. This sequence executes in 4 clock cycles (200ns).

A first-in-first-out (FIFO) buffer is used to match the speed of message transmission to the network as shown in Figure 13. In some cases, the MDP cannot send message words as fast as the network can transmit them. Without a buffer, *bubbles* (absence of words) would be injected into the network pipeline degrading performance. The SEND instruction loads one or two words into the buffer. When the message is complete or the eight-word buffer is full, the contents of the buffer are launched into the network.

Early in the design of the MDP we considered making a message send a single instruction that took a message template, filled in the template using the current addressing environment, and transmitted the message. Each template entry specified one word of the message as being either a constant, the contents of a data register, or a memory reference offset from an address register (like an operand descriptor). The template approach was abandoned in favor of the simpler one or two operand SEND instruction because the template did not significantly reduce code space or execution time. A two operand SEND instruction results in code that is nearly as dense as a template and can be implemented using the same control logic used for arithmetic and logical instructions.

Previous concurrent computers have used direct-memory access (DMA) or I/O channels to inject messages into the network. First an instruction sequence composed a message in memory. DMA registers or channel command words were then set up to initiate sending. Finally, the DMA

Mnemonic	Operands	Name	Op	Modes	Types
General Movement and Type Instructions					
READ	Src,Rd	Move Word	$01	R,A,m,i,c	All but CFUT
WRITE	Rs,Dst	Move Word	$02	m	All
READR	Src,Rd	Read Register	$03	Register	All but CFUT
WRITER	Rs,Dst	Write Register	$04	Register	All
RTAG	Src,Rd	Read Tag	$05	R,A,m,i,c	All but CFUT
WTAG	Rs,Src,Rd	Write Tag	$06	R,A,m,i,c	All,Int
LDIP	Src	Load IP	$07	R,A,m,i,c	Ip
LDIPR	Src	Load IP from Register	$08	Register	Ip
CHECK	Rs,Src,Rd	Check Tag	$09	R,A,m,i,c	All,Int
Arithmetic and Logic Instructions					
CARRY	Rs,Src,Rd	Carry from Add	$0A	R,A,m,i,c	Int,Int
ADD	Rs,Src,Rd	Add	$0B	R,A,m,i,c	Int,Int
SUB	Rs,Src,Rd	Subtract	$0C	R,A,m,i,c	Int,Int
MULH	Rs,Src,Rd	Multiply High	$0E	R,A,m,i,c	Int,Int
MUL	Rs,Src,Rd	Multiply	$0F	R,A,m,i,c	Int,Int
ASH	Rs,Src,Rd	Arithmetic Shift	$10	R,A,m,i,c	Int,Int
LSH	Rs,Src,Rd	Logical Shift	$11	R,A,m,i,c	Int,Int
ROT	Rs,Src,Rd	Rotate	$12	R,A,m,i,c	Int,Int
AND	Rs,Src,Rd	And	$18	R,A,m,i,c	Int,Int or Bool,Bool
OR	Rs,Src,Rd	Or	$19	R,A,m,i,c	Int,Int or Bool,Bool
XOR	Rs,Src,Rd	Xor	$1A	R,A,m,i,c	Int,Int or Bool,Bool
FFB	Src,Rd	Find First Bit	$1B	R,A,m,i,c	Int
NOT	Src,Rd	Not	$1C	R,A,m,i,c	Int or Bool
NEG	Src,Rd	Negate	$1D	R,A,m,i,c	Int
LT	Rs,Src,Rd	Less Than	$20	R,A,m,i,c	Int,Int or Bool,Bool
LE	Rs,Src,Rd	Less Than or Equal	$21	R,A,m,i,c	Int,Int or Bool,Bool
GE	Rs,Src,Rd	Greater Than or Equal	$22	R,A,m,i,c	Int,Int or Bool,Bool
GT	Rs,Src,Rd	Greater Than	$23	R,A,m,i,c	Int,Int or Bool,Bool
EQUAL	Rs,Src,Rd	Equal	$24	R,A,m,i,c	Int,Int or Bool,Bool or Sym,Sym
NEQUAL	Rs,Src,Rd	Not Equal	$25	R,A,m,i,c	Int,Int or Bool,Bool or Sym,Sym
EQ	Rs,Src,Rd	Pointer Equal	$26	R,A,m,i,c	All but CFut or Fut
NEQ	Rs,Src,Rd	Pointer not Equal	$27	R,A,m,i,c	All but CFut or Fut
Network Instructions					
SEND	Src,P	Send	$34	R,A,m,i,c	All but CFut
SENDE	Src,P	Send and End	$35	R,A,m,i,c	All but CFut
SEND2	Src,Rs,P	Send 2	$36	R,A,m,i,c	All but CFut
SEND2E	Src,Rs,P	Send 2 and End	$37	R,A,m,i,c	All but CFut
Associative Lookup Table Instructions					
XLATE	Rs,Dst,C	Associative Lookup	$28	R,A	All but CFut
ENTER	Src,Rs	Associative Enter	$29	R	All but CFut,All but CFut
PROBE	Rs,Dst	Probe Associative Cache	$2D	R	All but CFut
Special Instructions					
NOP		NOP	$00		
INVAL		Invalidate	$2A		
SUSPEND		Suspend	$30		
CALL	Src	System Call	$31	R,A,m,i	Int
Branches					
BR	Src	Branch	$38	R,I	Int
BNIL	Rs,Src	Branch if NIL	$3A	R,I	All but CFut,Int
BNNIL	Rs,Src	Branch if Non-NIL	$3B	R,I	All but CFut,Int
BF	Rs,Src	Branch if False	$3C	R,I	Bool,Int
BT	Rs,Src	Branch if True	$3D	R,I	Bool,Int
BZ	Rs,Src	Branch if Zero	$3E	R,I	Int,Int
BNZ	Rs,Src	Branch if NonZero	$3F	R,I	Int,Int

Figure 11: The MDP instruction set. Instructions are included to inject messages into the network and to enter and retrieve translations.

```
SEND    R0          ; send net address
SEND2   R1,R2       ; header and receiver
SEND2E  R3,[3,A3]   ; selector and continuation - end msg.
```

Figure 12: MDP assembly code to send a 4 word message uses three variants of the SEND instruction.

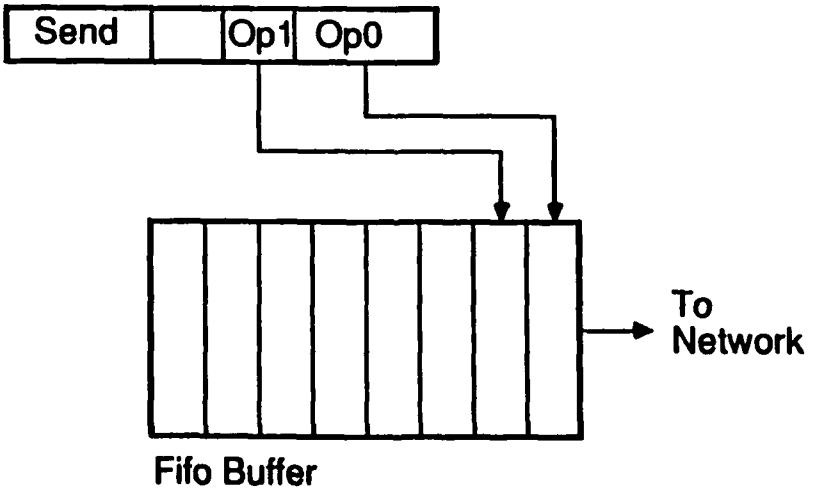

Figure 13: A FIFO buffer is used to match network speed. The SEND instruction loads message words into the buffer. When the message is complete or the buffer is full, the message is *launched.*

controller transferred the words from the memory into the network. This approach to message sending is too slow for two reasons. First, the entire message must be transferred across the memory interface twice, once to compose it in memory and a second time to transfer it into the network. Second, for very short messages, the time required to set up the DMA control registers or I/O channel command words often exceeds the time to simply send the message into the network.

4.3 Message Reception

The MDP maintains two message/scheduling queues (one for each priority level) in its on-chip memory. The queues are implemented as circular buffers. As shown in Figure 14a, the `QBM` (queue base and mask) register determines the location of this queue. The `QHL` (queue head and length) register holds the present state of the queue.

As messages arrive over the network, they are buffered in the appropriate queue. To improve memory bandwidth, messages are enqueued by rows (Figure 14b). Incoming message words are accumulated in a row buffer until the row buffer is filled or the message is complete. The row buffer is then written to memory. The head and length fields of the `QHL` register are added together to form the address of the queue tail and this address is *wrapped* by masking with the `QBM` register as shown in Figure 14c. After the row is written, the queue length is incremented by four.

It is important that the queue have sufficient performance to accept words from the network at the same rate at which they arrive. Otherwise, messages would backup into the network causing congestion. The queue row buffers in combination with hardware update of queue pointers allow enqueuing to proceed using one memory cycle for each four words received. Thus a program can execute in parallel with message reception with little loss of memory bandwidth.

Providing hardware support for allocation of memory in a circular buffer on a multicomputer is analogous to the support provided for allocation of memory in push-down stacks on a uniprocessor. Each message stored in the MDP message queue represents a method activation much as each stack frame allocated on a push-down stack represents a procedure activation.

An alternative queue organization, considered early in the MDP project, allocated storage from the heap for each incoming message. This eliminated the need to copy messages when a method suspended for intermediate results. However, the cost of allocating and reclaiming storage for each message proved to be prohibitive. Instead, we settled on the preallocated circular buffer. When a method suspends for intermediate results, message arguments are copied into a context object. The overhead of this copying is small since the context must be created anyway to specify a continuation and to hold live variables. The fixed buffer also provides a convenient layering. Priority zero messages are sent when the memory allocator runs out of room and priority one messages are sent when the priority zero queue fills.

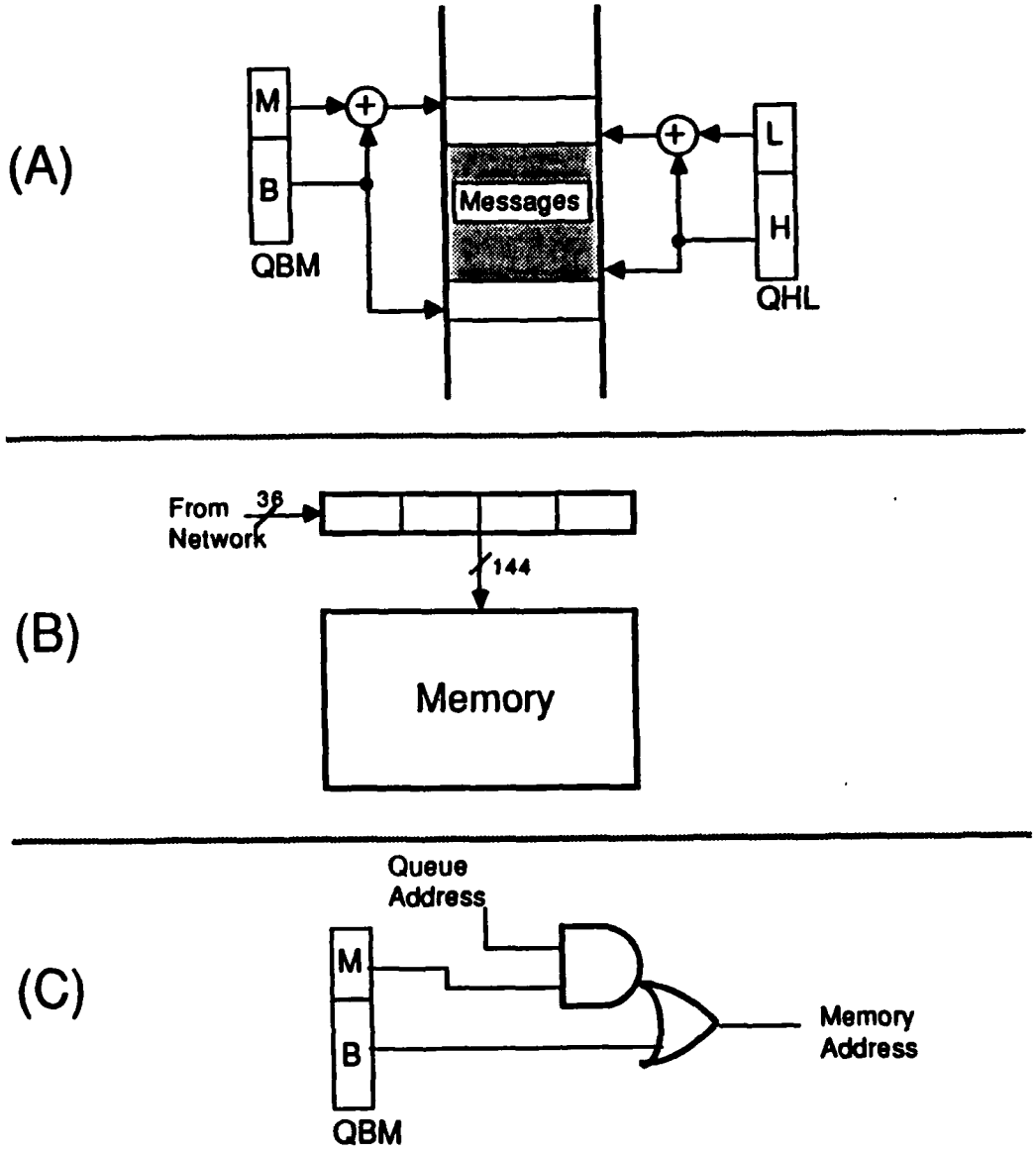

Figure 14: Message reception: (a) The QBM and QHL registers maintain a circular buffer message queue in local memory for each priority level. (b) Messages are enqueued a row (4-words) at a time to improve memory bandwidth. (c) Addresses are wrapped by masking with QBM.

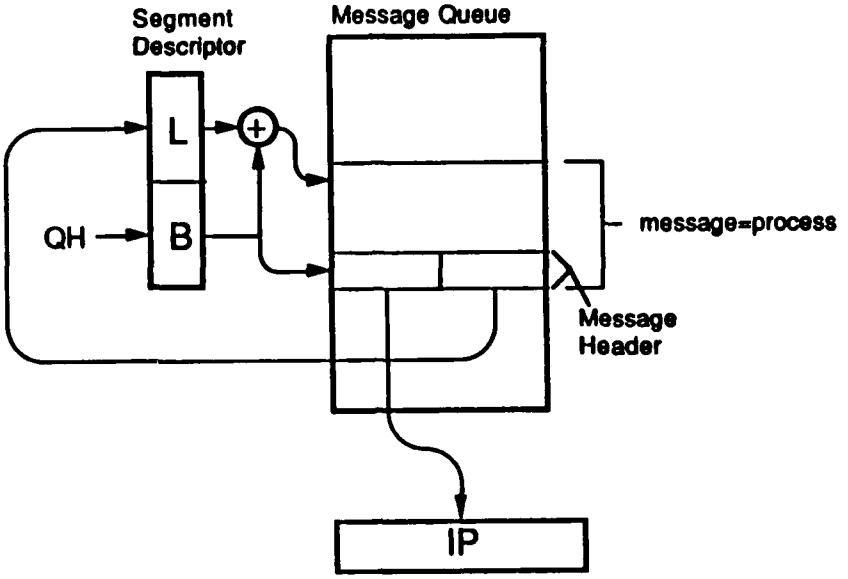

Figure 15: Message dispatch. In one clock cycle, a new task is created by (1) setting the IP to change the thread of control and (2) creating a message segment to provide the initial addressing environment.

4.4 Dispatch

Each message in the queues of an MDP represents a task that is ready to run. When the message reaches the head of the queue, a task is created to handle the message. At any time, the MDP is executing the task associated with the first message in the highest priority non-empty queue. If both queues are empty, the MDP is idle – viz., executing a background task. Sending a message implicitly schedules a task on the destination node. This simple two-priority scheduling mechanism removes the overhead associated with a software scheduler. More sophisticated scheduling policies may be implemented on top of this substrate.

Messages become *active* either by arriving while the node is idle or executing at a lower priority, or by being at the head of a queue when the preceding message *suspends* execution. When a message becomes active a task is created to handle it. Task creation, changing the thread of control and creating a new addressing environment, are performed in one clock cycle as shown in Figure 15. Every message header contains a message *opcode* and the message *length*. The message opcode is loaded into the IP to start a new thread of control. The length field is used along with the queue head to create a message segment descriptor in A3 that represents the initial addressing environment for the task. The message handler code may open additional segments by translating object IDs in the message into segment descriptors.

No state is saved when a task is created. If a task is preempting lower priority execution, it executes in a separate set of registers. If a task, A, becomes active when an earlier task, B, at the same priority suspends, B is responsible for saving its live state before suspending.

```
MOVE    [1,A3],R0   ; get method id
XLATE   R0,A0       ; translate to segment descriptor
LDIP    INITIAL_IP  ; transfer control to method
```

Figure 16: MDP assembly code for the CALL message.

The dispatch mechanism is used directly to process messages requiring low latency (e.g., combining and forwarding). Other messages (e.g., remote procedure call) specify a handler that locates the required method (using the translation mechanism described below) and then transfers control to it.

For example, a remote procedure call message is handled by the call handler code as shown in Figure 16. The execution of this handler is depicted in Figure 17. The first instruction gets the method ID (offset 1 into the message segment reference by A3). The next instruction translates this method ID into a segment descriptor for the method and places this descriptor in A0. If the translate faults, because the method is not resident or the descriptor is not in the translation cache, the fault handler *fixes* the problem and reschedules the message. If the translation succeeds, the final instruction transfers control to the method. The method code may then read in arguments from the message queue. The argument object identifiers are translated to physical memory base/length pairs using the translate instruction. If the method needs space to store local state, it may create a context object. When the method has finished execution, or when it needs to wait for a reply, it executes a SUSPEND instruction passing control to the next message.

An early version of the MDP had a fixed set of message handlers in microcode. An analysis of these handlers showed that their performance was limited by memory accesses. Thus there was little advantage in using microcode. The microcode was eliminated, the handlers were recoded in assembly language, and the *message opcode* was defined to be the physical address of the handler routine. Frequently used handlers are contained in an on-chip ROM. This approach simplifies the control structure of the machine and gives us flexibility to redefine message handlers to fix bugs, for instrumentation (e.g., to count the number of sends), and to implement new message types.

4.5 Synchronization with Tags

Every register and memory location in the MDP includes a 4-bit tag that indicates the type of data occupying the location. The MDP uses tags for synchronization on data availability in addition to their conventional uses for dynamic typing and run-time type checking. Two tags are provided for synchronization: `future`, and `c-future`. A `future` tag is used to identify a named placeholder for data that is not yet available [19]. Applying a strict operator to a `future` causes a fault. A `future` can, however, be copied without faulting. A `c-future` tag identifies a cell awaiting data. Applying any operator to a `c-future` causes a fault. As they are unnamed

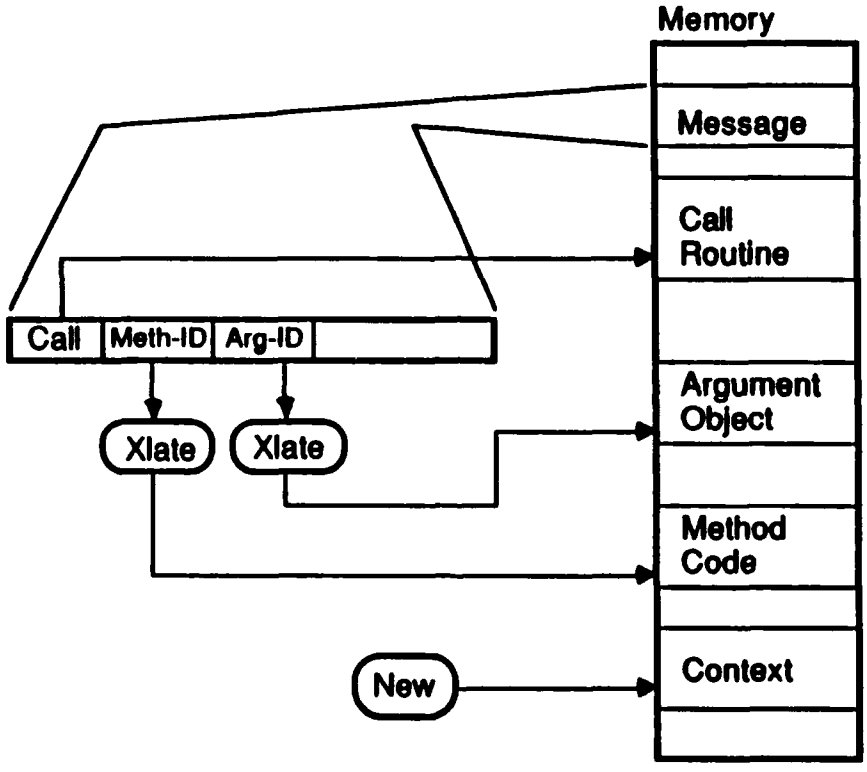

Figure 17: The CALL message invokes a method by translating the method identifier to find the code, creating a context (if necessary) to hold local state, and translating argument identifiers to locate arguments.

placeholders, they cannot be copied.

The c-future tag is used to suspend a task if it attempts to access data that has not yet arrived from a remote node. When a task sends a message requesting a reply, it marks the cell that will hold the reply as a c-future. Any attempt to reference the reply before it is available will fault and suspend the task. When the reply arrives, it overwrites the c-future and resumes the task if it was suspended. For example, when the task executing in Context 37 in Figure 2 sends the Sum message, it marks Slot 3 of its context as a c-future. The reply message overwrites Slot 3 to indicate data presence.

The future tag is used to implement named futures as in Multilisp [19]. Futures are more general than c-futures in that they can be copied. However, they are much more expensive than c-futures. A memory area and a name must be allocated for each future generated.

4.6 Translation

The MDP is an experiment in unifying shared-memory and message-passing parallel computers. Shared-memory machines provide a uniform global name space (address space) that allows processing elements to access data regardless of its location. Message-passing machines perform communication and synchronization via node-to-node messages. These two concepts are not mutually exclusive. The MDP provides a virtual addressing mechanism intended to support a global name space while using an execution mechanism based on message passing.

The MDP implements a global virtual address space using a general translation mechanism. The MDP memory allows both indexed and set-associative access. By building comparators into the column multiplexer of the on-chip RAM, we are able to provide set-associative access with only a small increase in the size of the RAM's peripheral circuitry.

The translation mechanism is exposed to the programmer with the ENTER and XLATE instructions. ENTER Ra,Rb associates the contents of Ra (the key) with the contents of Rb (the data). The association is made on the full 36 bits of the key so that tags may be used to distinguish different keys. XLATE Ra,Ab looks up the data associated with the contents of Ra and stores this data in Ab. The instruction faults if the lookup *misses* or if the data is not a segment descriptor. XLATE Ra,Rb can be used to lookup other types of data. This mechanism is used by our system code to cache ID to segment descriptor (virtual to physical) translations, to cache ID to node number (virtual to physical) translations, and to cache class/selector to segment descriptor (method lookup) translations.

Tags are an integral part of our addressing mechanism. An ID may translate into a segment descriptor for a local object, or a node address for a global object. The tag allows us to distinguish these two cases and a fault provides an efficient mechanism for the test. Tags also allow us to distinguish an ID key from a class/selector key with the same bit pattern.

Most computers provide a set associative cache to accelerate translations. We have taken this mechanism and exposed it in a pair of instructions that a systems programmer can use for any translation. Providing this general mechanism gives us the freedom to experiment with different address translation mechanisms and different uses of translation. We pay very little

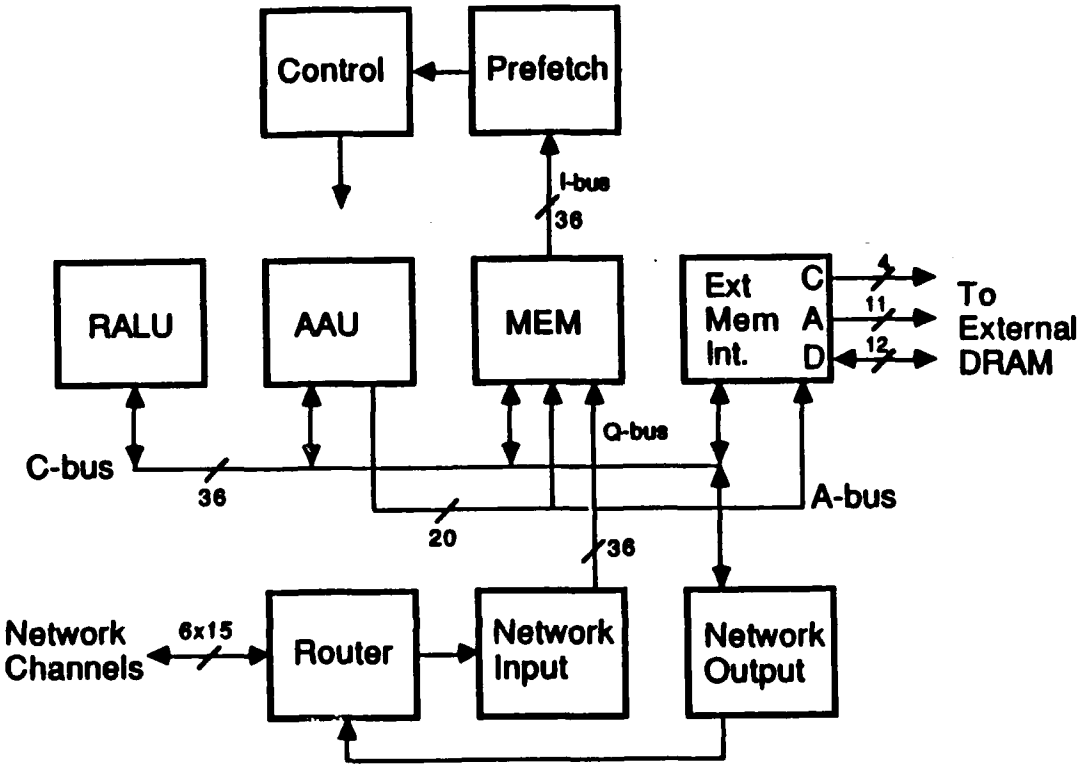

Figure 18: MDP block diagram.

for this flexibility since performance is limited by the number of memory accesses that must be performed.

4.7 Micro Architecture

The MDP consists of eight major subsystems as shown in Figure 18.

- **Control and Prefetch** The controller interprets an instruction sequence and monitors the state of the queues and network to generate state sequences that control the operation of the remaining blocks.
- **RALU** The general registers and ALU. The standard arithmetic, logical and comparison instructions are performed in this block.
- **AAU** The address arithmetic unit generates all memory addresses for data read and write, instruction fetches, network enqueuing, and task dispatch.
- **Memory** The memory block is a 4K x 36-bit static read/write memory. It includes input row buffers for enqueuing network data, one output row buffer for reading instructions, and comparators for implementing set associative access.

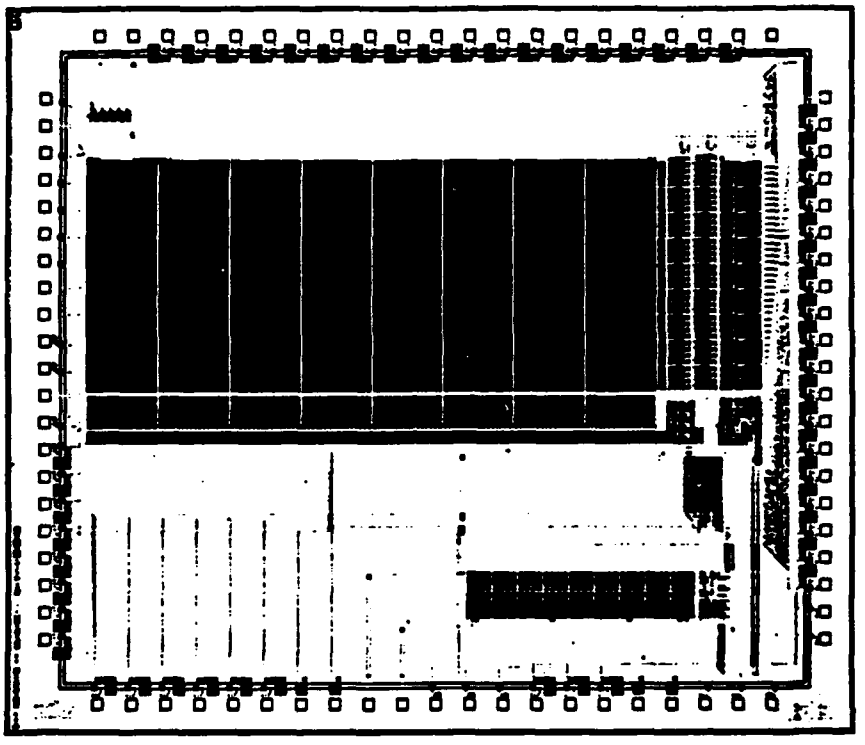

Figure 19: Photomicrograph of an MDP memory test chip.

- **External Memory Interface** This subsystem generates timing signals, multiplexed addresses, and error detection and correction to interface standard dynamic memory components to the MDP.
- **Network Input** The network input module accepts asynchronous data from the network, formats the 9-bit flits into 36-bit words, assembles words into the queue row buffers, and signals the control module to run an enqueue cycle to write the row buffer to memory.
- **Network Output** The network output contains a FIFO send buffer, and logic to sequence messages into the asynchronous network. This unit also subtracts the node address from the message header to convert absolute message addresses into relative addresses.
- **Router** The router is described in Section 3. It handles routing and flow control to deliver messages across the network.

An MDP memory test chip was implemented to test the feasibility of implementing row buffers and comparators in the peripheral circuitry of a memory[20]. The chip, shown in Figure 19, is a 1K-word $\times$ 37-bit (36 + parity) dynamic read/write memory organized as 256 rows of 148 bits. It is implemented in a 2μ double-metal CMOS technology. The circuitry at the bottom of the memory array includes one row buffer and the comparator and multiplexer circuitry required for set-associative access. This circuitry requires less than 10% of the total array area.

5 The Jellybean Operating System

JOSS is an operating system for the J-Machine that is designed to efficiently support fine-grain concurrent computation where tasks are very short (20 instructions), and data objects are very small (8 words) [36]. It is also tailored for an environment where local computation is inexpensive. Communication bandwidth and memory capacity are the limiting resources.

JOSS consists of a collection of system calls, fault handlers, message handlers, and remote-procedure call (RPC) code. System calls and fault handlers are similar to their counterparts in sequential systems. A message handler is a physically addressed system routine that is executed each time a particular type of message is received.

5.1 Abstractions

All JOSS abstractions are constructed from objects. System objects that are handled specially include the following:

- A *Method* is an object containing code. The system performs hierarchical distribution of methods and caches methods locally on each node.
- A *Context* is an object containing the state of a task. The system provides special allocation and deallocation of contexts to speed task creation and provides services for suspending and resuming contexts.
- A *Class* is an object that defines the properties of a specific class (or type) of object.

All computation takes place by sending messages between objects. Consider, for example, sending the message `increment` to the object `counter`. The *class* of `counter` is accessed to look up a *method* to be executed in response to an `increment` message. A task is then created to execute the code in this method. If the task must suspend its execution to await a message, it saves its state in a *context* object before relinquishing the processor.

5.2 Global Object Namespace

Most message-passing multicomputers have a separate memory address space on each node. Nodes interact only by sending messages between processes [35]. A partitioned address space makes it difficult to construct distributed data structures [5], limits the size of a processes address space to the memory size of a node, and requires entire processes to be relocated to balance memory use. Also, because storage on remote nodes cannot be directly accessed, these machines replicate the operating system and application code on each node.

JOSS overcomes these limitations by providing a global object namespace. All data and code are stored in objects. Each object is assigned a unique global ID. Given an object ID, a task on any node can reference the corresponding object. Objects are free to migrate between

nodes. Accesses to objects are bounds checked and protected. The system supports distributed objects [16]. Large distributed objects are implemented as a collection of small constituent objects accessed via a single ID. A one to many translation service prevents the single ID from becoming a bottleneck.

A global object namespace provides many of the advantages of a shared memory multicomputer while retaining the scalability of a message-passing machine. Distributed data structures are easily constructed by linking together objects on different nodes using IDs. Processes have an address space limited only by the size of an ID. Also, code need not be replicated on each node since it can be referenced through its ID.

To support this global object space, JOSS provides (1) services to allocate and deallocate objects, and (2) services to translate object names (IDs) into object locations. Both functions are layered with one component providing the service locally within a node and a second component extending the service across the network.

Objects are created locally by allocating a contiguous region of memory off the top of the heap and assigning a unique name (ID) to the object. The global ID space is partitioned so that nodes may assign unique global IDs autonomously. Object creation is extended across the network by providing a `NEW` message that creates an object of a specified class on a remote node and returns its ID.

Objects are deleted by marking. Their storage is reclaimed by a compactor that copies objects down in memory to fill unused holes. As segments are relocated during compaction, the local translation table is updated and all segment registers are invalidated. Compaction is very fast because local memory is small and fast and because the operation is completely local – no communication is required.

Given an object ID, an object is located in two steps. First, a distributed global name table is accessed to find the node on which the object is resident. A message is then sent to the node where the ID is translated into a segment descriptor for the object. The segment descriptor for an object is strictly local information. Thus, each node may relocate objects locally without interacting with any other nodes.

Accessing the global name table involves a message send. To avoid this indirection, nodes may maintain hints as to the present location of a remote object. The global name table must be consulted, however, if the hint becomes stale or is discarded to free up space.

In a fine-grain multicomputer, segment-based memory management is preferred to paging because fine-grain relocation and protection is required. The ability to compact all of memory in a few milliseconds eliminates concerns over external fragmentation. Internal fragmentation is an issue. Objects are small and must be protected and relocated individually. To support fine-grain computation, a paging system would either have to have a very small page size (8 words), or sacrifice protection by packing unrelated objects into the same page.

5.3 Task Management

To make use of a machine with tens of thousands of processing nodes one must divide the problem at hand into many small tasks. A typical fine-grain task executes only 20 instructions before exiting or suspending to await a message. Conventional operating systems are poorly suited to manage such fine-grain tasks. They require thousands of instructions to create a task, and hundreds of instructions to context switch between tasks.

JOSS provides a set of low-overhead task management services that efficiently support fine-grain tasks. Using the hardware task scheduling and dispatching mechanisms of the MDP, creating a task, suspending a task, resuming a task, and destroying a task each require fewer than ten instructions. This inexpensive task management is provided without sacrificing protection. Each task executes in its own addressing environment.

Context allocation is much faster than object allocation. Fixed size contexts are allocated off a list of free contexts. After waiting for all pending replies, the context storage and name are recycled by appending them to the list of free contexts. A strict request/reply protocol insures that there are no dangling references to a recycled context name.

Low overhead task switching in JOSS depends on three features of the design. (1) The MDP dispatch mechanism to eliminates scheduler overhead. (2) Contexts to hold the state of a task are allocated quickly. (3) The small MDP register set limits the state to be saved on suspension to five words.

5.4 Input/Output

No special support is provided in JOSS for input and output. I/O devices are considered to be non-relocatable objects. These I/O objects respond to messages to transfer information in and out of the system. I/O devices are protected by restricting distribution of their object names. A disk object, for example, may be protected by allowing it to be referenced only through file objects.

6 Conclusion

The J-Machine is a general purpose parallel computer. It provides general mechanisms for communication, synchronization, and translation rather than hardwiring mechanisms for a specific model of computation. These mechanisms efficiently support many proposed models of computation. Using these mechanisms, the overhead of creating a task on a remote node is reduced to a few microseconds. This low overhead permits concurrency to be exploited at a fine-grain size.

The J-Machine provides a substrate on which actor systems can be built. Creating an actor on a remote node, sending a message to an actor to request a service, and dispatching an actor to execute a script in response to a message can all be performed with a few microseconds overhead.

The global address space of the machine can be used to name actors and continuations.

The J-Machine is designed to make efficient use of silicon and wiring area. Each message driven processing node is a *jellybean* part. It can be fabricated in the same technology used to manufacture existing commodity semiconductor parts such as DRAMs. The network is designed to make efficient use of wires so the machine can be packaged densely – with processing nodes consuming most of the volume. There are no large wiring channels.

At the time of this writing (September 1988), the project is currently in the advanced design stage. Message-level, instruction-level, and register transfer level simulators have been built to test the J-Machine design. Prototype versions of JOSS and the CST compiler are operational. Gate and transistor level schematics are in the process of being drawn. We expect to complete the processing node chip design in late 1989 and have a prototype J-Machine System running in mid 1990.

Acknowledgement

The following MIT students have contributed to the work described here: Linda Chao, Andrew Chien, Stuart Fiske, Soha Hassoun, Waldemar Horwat, Michael Larivee, Rich Lethin, John Keen, Peter Nuth, Paul Song, Brian Totty, and Scott Wills.

I thank Tom Knight, Gerry Sussman, Steve Ward, Dave Gifford, Tom Leighton, and Carl Hewitt of MIT, Chuck Seitz and Bill Athas of Caltech, and Paul Carrick, Greg Fyler, Mark Vestrich, Justin Rattner, and George Cox of Intel Corporation for many valuable suggestions, comments, and advice.

References

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] Ametek Computer Research Division, *Series 2010 Product Description*, 1987.

[3] Athas, W.C., and Seitz, C.L., *Cantor Language Report*, Technical Report 5232:TR:86, Dept. of Computer Science, California Institute of Technology, 1986.

[4] BBN Advanced Computers, Inc., *Butterfly Parallel Processor Overview*, BBN Report No. 6148, March 1986.

[5] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer, Hingham, MA, 1987.

[6] Dally, William J. and Seitz, Charles L., "The Torus Routing Chip," *J. Distributed Systems*, Vol. 1, No. 3, 1986, pp. 187-196.

[7] Dally, William J. "Wire Efficient VLSI Multiprocessor Communication Networks," *Proceedings Stanford Conference on Advanced Research in VLSI,* Paul Losleben, Ed., MIT Press, Cambridge, MA, March 1987, pp. 391-415.

[8] Dally, William J. and Seitz, Charles L., " Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers,* Vol. C-36, No. 5, May 1987, pp. 547-553.

[9] Dally, William J. et.al., "Architecture of a Message-Driven Processor," *Proceedings of the* 14^{th} *ACM/IEEE Symposium on Computer Architecture,* June 1987, pp. 189-196..

[10] Dally, William J., and Song, Paul., "Design of a Self-Timed VLSI Multicomputer Communication Controller," *Proc. International Conference on Computer Design, ICCD-87,* 1987, pp. 230-234.

[11] Dally, William J., "Concurrent Data Structures," Chapter 7 in *Message-Passing Concurrent Computers: Their Architecture and Programming,* C.L. Seitz et. al., Addison-Wesley, Reading, MA, publication expected 1988.

[12] Dally, William J., "Concurrent Computer Architecture," *Proceedings of Symposium on Parallel Computations and Their Impact on Mechanics,* 1987.

[13] Dally, William J. et.al., *Message-Driven Processor Architecture, Version 11* MIT Artificial Intelligence Laboratory Memo No. 1069, August, 1988.

[14] Dally, William J. "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Transactions on Computers,* to appear.

[15] Dally, W.J., "Fine-Grain Concurrent Computers", *Proc.* 3^{rd} *Symposium on Hypercube Concurrent Computers and Applications,* ACM 1988.

[16] Dally, W.J., and Chien A.A., "Object Oriented Concurrent Programming in CST," *Proc.* 3^{rd} *Symposium on Hypercube Concurrent Computers and Applications,* ACM 1988.

[17] Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer,* Vol. 13, No. 11, Nov. 1980, pp. 48-56.

[18] Flaig, Charles, M., *VLSI Mesh Routing Systems,* Technical Report 5241:TR:87, Dept. of Computer Science, California Institute of Technology, 1987.

[19] Halstead, Robert H., "Parallel Symbolic Computation," *IEEE Computer,* Vol. 19, No. 8, Aug. 1986, pp. 35-43.

[20] Hassoun, S. *Memory Design for a Message-Driven Processor,* MIT VLSI Memo, June 1988.

[21] Hoare, C.A.R., "Communicating Sequential Processes," *Comm. ACM,* Vol. 21, No. 8, August 1978, pp. 666-677.

[22] Inmos Limited, *IMS T424 Reference Manual,* Order No. 72 TRN 006 00, Bristol, United Kingdom, November 1984.

[23] Intel Scientific Computers, *iPSC User's Guide*, Order No. 175455-001, Santa Clara, CA, Aug. 1985.

[24] Kermani, Parviz and Kleinrock, Leonard, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Vol 3., 1979, pp. 267-286.

[25] Knight, Tom, and Krymm, Alex, "Self Terminating Low-Voltage Swing CMOS Output Driver," *Proc. Custom Integrated Circuits Conference*, 1987.

[26] Ligtenberg, Adriaan, Presentation at *1987 Princeton Workshop on Algorithm, Architecture, and Technology Issues in Models of Concurrent Computation*, October 1987.

[27] Lutz, C., et. al., "Design of the Mosaic Element," *Proc. MIT Conference on Advanced Research in VLSI*, Artech Books, 1984, pp. 1-10.

[28] Mead, Carver A. and Conway, Lynn A., *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.

[29] Palmer, John F., "The NCUBE Family of Parallel Supercomputers," *Proc. IEEE International Conference on Computer Design, ICCD-86*, 1986, p. 107.

[30] Pfister, G.F. et. al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. International Conference on Parallel Processing, ICPP*, 1985, pp. 764–771.

[31] Seitz, Charles L., "System Timing" in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Ch. 7.

[32] Seitz, Charles L., et al., *The Hypercube Communications Chip*, Display File 5182:DF:85, Dept. of Computer Science, California Institute of Technology, March 1985.

[33] Seitz, Charles L., "The Cosmic Cube", *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.

[34] Seitz, Charles L., Athas, William C., Dally, William J., Faucette, Reese, Martin, Alain J. , Mattisson, Sven, Steele, Craig S., and Su, Wen-King, *Message-Passing Concurrent Computers: Their Architecture and Programming*, Addison-Wesley, publication expected 1988.

[35] Su, Wen-King, Faucette, Reese, and Seitz, Charles L., *C Programmer's Guide to the Cosmic Cube*, Technical Report 5203:TR:85, Dept. of Computer Science, California Institute of Technology, September 1985.

[36] Totty, Brian, *An Operating Environment for the Jellybean Machine*, MIT Artificial Intelligence Laboratory Memo No. 1070, 1988.